SANDIA REPORT

SAND98-2102
Unlimited Release
Printed September 1998

An Automatic Coastline Detector for Use with SAR Images

Ireena A. Erteza

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615)576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND98-2102
Unlimited Release
Printed September 1998

An Automatic Coastline Detector for Use with SAR Images

Ireena A. Erteza
Signal Processing and Research Department
Sandia National Laboratories
P.O. Box 5800 Albuquerque, NM 87185-1207

Abstract

SAR imagery for coastline detection has many potential advantages over conventional optical stereoscopic techniques. For example, SAR does not have restrictions on being collected during daylight or when there is no cloud cover. In addition, the techniques for coastline detection with SAR images can be automated.

In this paper, we present the algorithmic development of an automatic coastline detector for use with SAR imagery. Three main algorithms comprise the automatic coastline detection algorithm. The first algorithm considers the image pre-processing steps that must occur on the original image in order to accentuate the land/water boundary. The second algorithm automatically follows along the accentuated land/water boundary and produces a single-pixel-wide coastline. The third algorithm identifies islands and marks them.

This report describes in detail the development of these three algorithms. Examples of imagery are used throughout the paper to illustrate the various steps in algorithms. Actual code is included in appendices. The algorithms presented are preliminary versions that can be applied to automatic coastline detection in SAR imagery. There are many variations and additions to the algorithms that can be made to improve robustness and automation, as required by a particular application.

Acknowledgements

This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

Contents

List of Figures

1 Introduction

Currently there are many miles of US coastline that have not been mapped. While adequate mapping is certainly of use in navigation, these maps are also of critical importance in determining US coastal boundaries. The locations of these boundaries are used in determining legal rights (ownership and liability) in various off-shore ventures.

Traditionally, coastline mapping has been done via optical stereoscopic imaging. The actual boundary is determined by some combination of the lowest low-tide and the highest high-tide. However, since optical stereoscopic images are used, it is often impossible to get images of the desired location at exactly lowest low-tide or highest high-tide. The optical stereoscopic imaging systems require the correct illumination and a clear view (i.e. no clouds). Optical stereoscopic imaging for coastline detection is also a time consuming, non-automated process.

In an effort to find more robust alternatives for coastline detection, SAR has been proposed. SAR has a variety of possible benefits. First, SAR does not require flying during daylight. Also, SAR is effective, even if clouds are present. In addition to forming a traditional SAR image, if interferometric SAR is used, it may be possible to form a height map of the area and to extract the boundary from the height map.

In this report, we will present an algorithm for automatically identifying the coastline boundary in a standard (non-interferometric) SAR image. Using this technique with any SAR image, a land/water boundary can be identified. That boundary, itself, could be a viable final result. However, if two SAR images are available at low and high tides, the coastal boundary could be found by judicious combination of the two high- and low-tide boundaries.

The main benefit of this coastline detection is that the process can be automated. The algorithm appears to be fairly robust, but it has only been tested on Alaska coastline data. This report describes in detail the development of the coastline algorithm and the parameters used. Examples of actual imagery are used throughout the paper to illustrate the various steps in algorithm. Actual code is included in appendices.

2 General Approach

The approach taken to solve the automatic coastline detection problem can be broken into three main parts. The first part is to perform a variety of image processing steps on the original image in order to accentuate the land/water boundary. This is necessary in order to make a robust contour following program that can work consistently on a wide variety of coastline data. The second part of automatic coastline detection is to automatically follow along the accentuated land/water

boundary resulting from the first step and to produce a single-pixel-wide boundary of the shoreline. The third part is to identify any existing "islands" in the image. An "island" is a group of pixels that meet the land criteria, but that are in what the algorithm identifies as water. At this time, the algorithm only identifies islands. No attempt is made to find coastlines of the islands, although this could easily be added.

2.1 Land/Water Boundary Enhancement

In order to be able to accentuate and identify a land/water boundary, one must come up with some criteria to distinguish land from water. Ultimately, we are interested in a single pixel boundary to mark the transition between land and water, therefore our criteria will be applied ultimately on a pixel by pixel basis. However, there is a large degree of pixel-to-pixel variety in a SAR image of either just land or just water. We must come up with a way of minimizing that variation within pixels of land and within pixels of water, while maintaining distinct characteristics for each. We achieve these goals through a series of image processing steps.

2.1.1 MEDIAN FILTERING

The first step is a median filtering operation. In this step a 3x3 window is scanned over the entire image. At each step in the scan, the center pixel is replaced by the median value of the 9 pixels in the window. The effect of a median filter is to remove single points whose values are out of line with neighboring pixels. In the particular context of SAR images of coastlines, a median filter minimizes (1) the speckle normally associated with SAR images and (2) the bright returns from ice, rough water or other matter in a predominantly dark part of the image (water).

The median filtering is actually performed twice, i.e. the output of the median filtering process described above is again median filtered. With each additional pass, there is added minimization in variation. Ultimately, however, the incremental change between passes becomes negligible. The median filtering process is time consuming on large images. Empirically two passes of a 3x3 median filter appear adequate for our purposes. This is discussed in more detail in the Implementation section.

A median filter can be thought of as a smoothing filter, but a significant difference with respect to edges must be understood and emphasized. A median filter will not blur the transition point or edge as a traditional, low-pass, smoothing filter would. This is important when maintaining a sharp and accurate transition point is required.

2.1.2 HISTOGRAM EQUALIZATION (FLATTENING)

The next step to accentuate the land/water boundary is a histogram flattening step. In this step, a histogram is made of the image after median filtering. The histogram is a count of how many times each pixel value (0 - 255) actually occurs in an image. The flattening step re-maps the pixel values in the image, such that the histogram after equalization is constant (hence the term "histogram flattening"). With a flat histogram, each possible pixel value occurs the same number of times, i.e. the probability density function for pixel values 0 through 255 is uniform.

The main effect of histogram equalization is to increase the contrast throughout the image. This is done by effectively allocating more pixel levels where the most pixels are originally, and allocating fewer pixel levels where there are fewer pixels originally.

An added benefit of histogram flattening is that it helps to "standardize" the look of the images. This will enable us to determine characteristics for land and water that are the same for different images. This feature is of particular importance in making a robust automatic coastline detector.

2.1.3 THRESHHOLDING

The next image processing step for land/water boundary enhancement is thresh-holding. This step is where we apply a distinction between land and water. In general, we expect the radar return from land to be higher than from water. As a result, land pixels generally have a higher pixel value than water pixels. There are exceptions, however. Ice or turbulence in the water can have a significant radar return (i.e. higher pixel values). Similarly, land that is in a shadow (from a cliff) can have a very low radar return, since it is not illuminated with much radar energy. In this step, we apply a threshhold of 200 to distinguish land from water in the histogram flattened image. This is a non-linear step, in which a pixel with a value < 200 is assigned a value of 0, and pixels with a value >200 do not have their pixel value changed.

Because of the histogram flattening step, the threshhold value of 200 should be effective for a variety of images. It is possible, however, that different imaging scenarios may produce SAR images that have different characteristics for land vs. water. In that case, this threshhold may need to be altered to optimize the algorithm's performance.

2.1.4 MAXIMUM FILTERING (DILATION)

The final image processing step for the land/water boundary enhancement is two passes through a maximum (or dilation) filter. The mechanics of this step are very similar to the median filtering step. In this case a window is scanned over the

entire image. However, at each step, the center pixel is replaced by the maximum value of all the pixels in the window.

The effect of a maximum filter is to make the brighter areas larger, and the darker areas smaller. Although the resulting image can be greatly distorted, the general shape of large areas of brightness and darkness are preserved. As a result, this step is useful in image segmentation. For our particular application, we would specifically like the dilation operator to accomplish two things: (1) Make a continuous exterior boundary (without any gaps) between interior land and exterior water. (2) Minimize shadow regions in the interior land.

The dilation filtering is performed with a 7x7 window. It is also performed in two passes. The 7x7 dilation process is time consuming for a large image. Two passes seems to be a nice compromise in terms of closing the exterior boundary gaps, while also not extending the actual borders too much. (With two 7x7 passes, it is possible to extend the land/water boundary by 14 pixels beyond the actual boundary.) If this extension is unacceptable, some additional steps may be added to mitigate the extension on the boundary, while still maintaining the benefit of closing gaps in the boundary. Closing the gaps in the boundary is critical to the operation of the automatic contour following algorithm, as discussed in the following section.

2.2 Contour Following

Once the image processing steps have been performed to enhance the land/water boundary, the next step is to form and mark a one pixel wide boundary between the land and water.

The algorithm we employ to do the contouring is based on one of the most simple available. The simple clockwise contour following algorithm by Duda and Hart is described in several places. [BB82, DH73] The technique involves scanning across an image until a pixel that is part of the object to be outlined is encountered. This is the starting point. From the pixel, turn left and move one pixel. Each time a pixel that is part of the object is encountered, turn left and move one pixel. If a non-object pixel is encountered, turn right and move one pixel. Repeating this process, the entire object boundary will be traversed (albeit in a serpentine fashion) in a clockwise direction. This is illustrated in Figure 1.

We extend and modify this algorithm to coastline detection as follows. For coastline detection, we do not necessarily have land that we completely encircle. In general, in an image of a coastline there is a main land mass on one side or another of the picture (right or left). If we always start our scans at the bottom of the image, it may be necessary to follow the contour in either a clockwise or counter-clockwise direction. We can extend the contour following algorithm from Duda and Hart to counter-clockwise following by: (1) turning *right* and moving one pixel if you encounter an object pixel; and (2) turning *left* and moving one

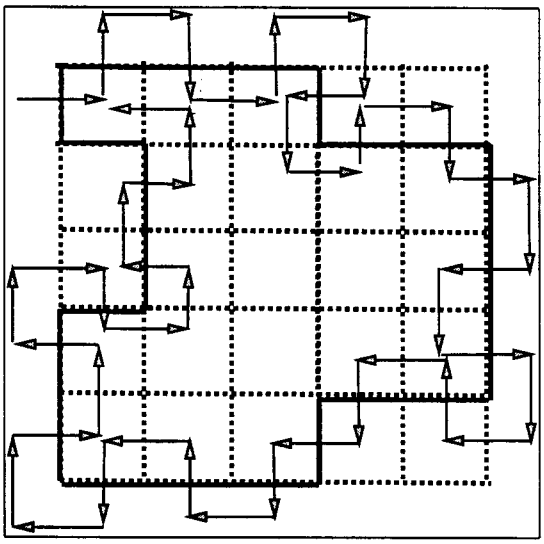

Figure 1: Example of how the contour of an object is traced in a clockwise direction.

pixel if you encounter a non-object pixel.

Therefore our coastline following algorithm will require the user to specify if land is on the right side or left side of the figure. The image then will be scanned from the bottom until an object pixel is reached. If land is on the right, the scan starts from the left, and vice versa. From this starting pixel, the boundary is traced clockwise if the land is on the right. It is traced counter-clockwise if the land is on the left. The coastline tracing will terminate when it reaches within the top 10 rows of the image, or if it returns to within one pixel or to the same row as the original starting pixel.

As the algorithm is implemented now, the program will terminate after it traces out a boundary island at the bottom of the image. Additional programming can be added to have the program continue until the main coastline has been found and traced.

The simple algorithm we are using has some drawbacks. First, it is possible for the algorithm to miss a pixel that is only connected diagonally with a previously identified part of the object. Also, this algorithm requires that the object to be outlined have no gaps in it. If there is a gap, the contour following algorithm will follow into the interior of the object. It is this requirement that necessitates the dilation step in the image preprocessing. In general, we expect that the two 7x7 dilation operations should remove gaps in the exterior boundary. There may be some cases, however, where this may not be sufficient. Additional dilation steps may be needed in some cases.

Finally, we don't want the serpentine path to be the boundary, so we mark any land pixels that the serpentine path contains as boundary pixels. A land pixel is any pixel with value >200. The boundary pixels are given a value of 255.

2.3 Island Detection and Marking

The final part of our automatic coastline detection is not actually involved with detecting the coastline. Instead, the purpose of this final step is to mark land pixels that are beyond the mainland, in water. The reason for this is that there may be barrier islands of substantial size, whose boundaries might actually be the coastline of interest. The way the automatic coastline detection algorithm is designed and implemented, it will in general find the coastline between the mainland and ocean. (The exception is for a barrier island that exists at the bottom of the image).

The goal of this final step is to mark islands, so that an analyst can determine if those barrier islands significantly alter the identified mainland coastline. At this time, this determination is made by an analyst. It should be possible, however, to automatically find the boundaries for the "islands", and to modify a "maximum coastline" appropriately.

3 Algorithm Implementation

3.1 Image Pre-Processing for Land/Water Boundary Enhancement

The image processing steps described in the previous section for land/water boundary enhancement are straightforward image processing steps. In order to implement them, we used a standard C language library for digital signal and image processing. The library routines used are found in *C Language Algorithms for Digital Signal Processsing* by Paul Embree and Bruce Kimble. [EK91] Additional routines for adding and removing the DSP header format, for threshholding and for doing nonlinear filtering with various size windows were written. The routines used in this project are included in the appendix.

3.2 Contour Following

The basic algorithm for contour following was described in the previous section. In order to implement both clockwise and counter-clockwise boundary following, without having to write two completely separate branches, requires some analysis.

3.2.1 Conventions

In order to do the analysis to combine the two cases for clockwise and counter-clockwise boundary following, we adopt some conventions and terminology. First, we assume that an image is indexed such that the upper left corner is (0,0). The

index for rows, i, increases as you go down. The index for columns, j, increases as you go right. This is illustrated in Figure 2.

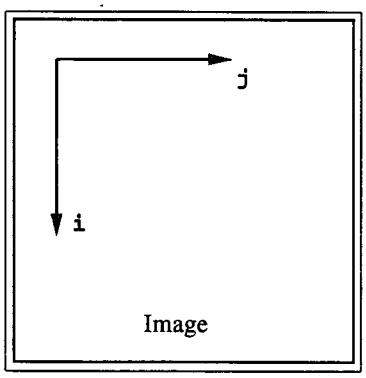

Figure 2: Convention for row and column indexing of an image.

Next, the orientation when entering a pixel is important. If a pixel is entered from the left, we label that verbally "IL" and numerically "0". (The vector describing entering from the left, $\rightarrow$, makes a 0° angle with respect to the positive x axis.) Similarly, if a pixel is entered from the right, we use the labels "IR" and "180". If a pixel is entered from the top, the verbal label is "IT" and "270". Finally, if a pixel is entered from the bottom, we use the labels "IB" and "90".

The final important piece of information required by the algorithm is if the boundary is to be traversed in a clockwise or counter-clockwise direction. We assign a numerical label of 1 for clockwise traversal, and -1 for counter-clockwise traversal.

With these conventions, we can do the analysis of the algorithms and write logic to handle the different steps required for clockwise and counter-clockwise traversal.

3.2.2 Starting point

When the first land pixel is identified, the start orientation and the index changes needed to get to the start pixel will be different for clockwise (CW) and counter-clockwise (CCW) traversal.

	CW (DIR = 1)	CCW (DIR = -1)
Orientation	180°	0°
Start Pixel	(i, j - 1)	(i, j + 1)

This can be reduced in both cases to the rules in the following table.

Orientation	90° + 90°* DIR
Start Pixel	(i, j - DIR)

3.2.3 Next Pixel Location

The location for the next pixel depends on 3 things: (1) the direction for boundary traversal; (2) orientation into current pixel; and (3) the current pixel value. This is illustrated in the following pictures and tables.

Case	Pixel Value = Land	Next Pixel Location	
		CW (DIR = 1)	CCW (DIR = -1)
"0" → □	Y	i - 1	i + 1
"0" → □	N	i + 1	i - 1
"270" ↓ □	Y	j + 1	j - 1
"270" ↓ □	N	j - 1	j + 1
"180" □ ←	Y	i + 1	i - 1
"180" □ ←	N	i - 1	i + 1
"90" □ ↑	Y	j - 1	j + 1
"90" □ ↑	N	j + 1	j - 1

For both CW and CCW traversal, this can be reduced as follows:

Case	Pixel Value = Land	Next Pixel Location
"0" → □	Y	i - DIR
"0" → □	N	i + DIR
"270" ↓ □	Y	j + DIR
"270" ↓ □	N	j - DIR
"180" □ ←	Y	i + DIR
"180" □ ←	N	i - DIR
"90" □ ↑	Y	j - DIR
"90" □ ↑	N	j + DIR

3.2.4 Orientation for Next Pixel

The orientation for the next pixel also depends on three things, as illustrated in the following table.

Current Orientation	Clockwise		Counter Clockwise	
	Land	Not Land	Land	Not Land
IL "0" → □	IB 180	IT 270	IT 270	IB 180
IT "270" ↓ □	IL 0	IR 180	IR 180	IL 0
IR "180" □ ←	IT 270	IB 90	IB 90	IT 270
IB "90" □ ↑	IR 180	IL 0	IL 0	IR 180

Once again, this can be reduced to the following logic for CW and CCW traversal.

Current Orientation	Orientation for Next Pixel	
	Land	Not Land
IL "0" → □	0+90*DIR	0-90*DIR
IT "270" ↓ □	270+90*DIR	270-90*DIR
IR "180" □ ←	180+90*DIR	180-90*DIR
IB "90" □ ↑	90+90*DIR	90-90*DIR

3.3 Island Detection and Marking

The goals for island detecting and marking seem quite simple, however it is somewhat complicated to implement. The way the island detection goal is achieved is by scanning the image and cleaning land pixels which are part of the "mainland". Intuitively, it is easy to just clear pixels from the right edge (assuming land is on right) until a boundary is reached. When a boundary is reached, the algorithm should switch from clearing pixels to saving land pixels. The factors which cause complications are bays or coves, and horizontal boundaries. With some analysis, however, logic can be written to handle these situations.

In general, when a single boundary pixel (i.e. it is not part of a number of consecutive horizontal boundary pixels) is encountered, the algorithm should toggle between clearing pixels and not clearing pixels. The case is not so straightforward, however, when a row of consecutive, horizontal boundary pixels are encountered. The analysis becomes more clear if we adopt a few more conventions.

3.3.1 Conventions

The topology at the beginning and end of a horizontal boundary run, along with the initial state of clearing/not clearing pixels and the direction of the scan will determine if the clear pixel variable should be toggled or not.

Let B designate a border pixel that is at the beginning of a horizontal run. Let E designate a border pixel that is at the end of a horizontal run. There are four possible topologies we need to label. These are illustrated in Figure 3. With either B or E in the center of a 3x3 window, if the pixels above and to the above right are also boundary pixels, the topology is labeled "1". (Note that the pixels in the first quadrant are boundary pixels.) If the pixels below and below right are also boundary pixels, the topology is labeled "4". (Note that the pixels in the fourth quadrant are boundary pixels.) Similarly, if the pixels above and above left are also boundary pixels, the topology is labeled "2". (Note that the pixels in the second quadrant are boundary pixels.) Finally, if the pixels below and below left are also boundary pixels, the topology is labeled "3". (Note that the pixels in the third quadrant are boundary pixels.)

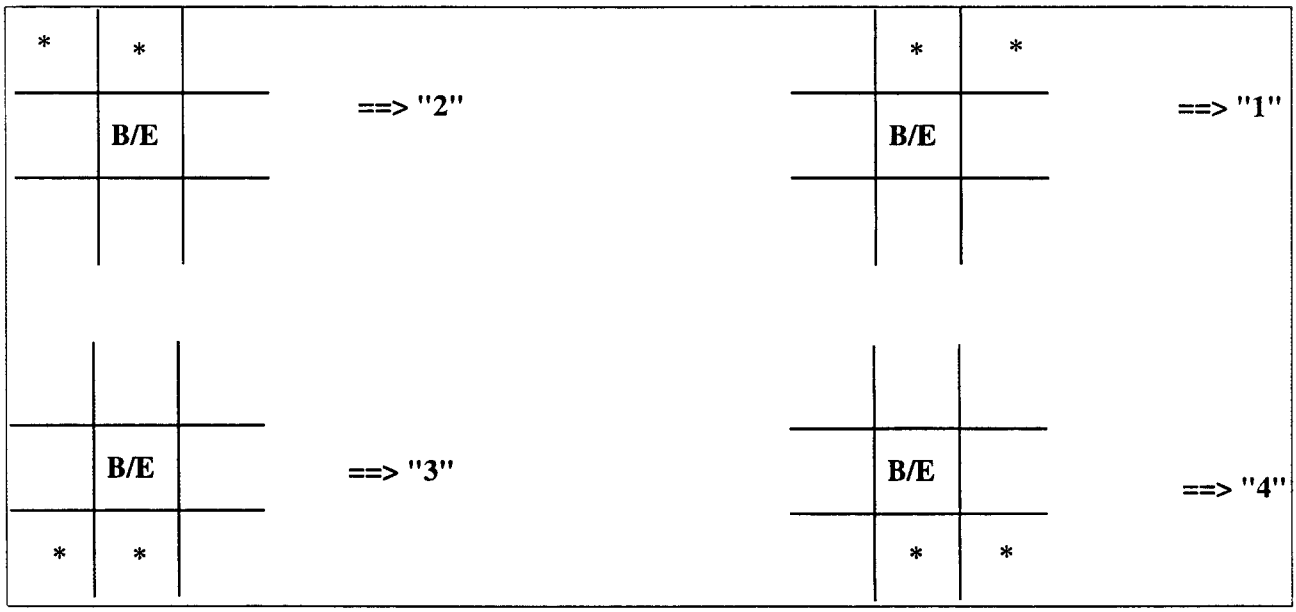

Figure 3: Convention for how beginning and ends of horizontal boundary runs are labeled. * indicates a boundary pixel.

The 8 cases for scanning from right to left are illustrated in Figures 4 and 5. In these figures, C indicates the clearing pixels state (i.e. pixels should be cleared), and $\overline{C}$ indicates the state where pixels aren't cleared.

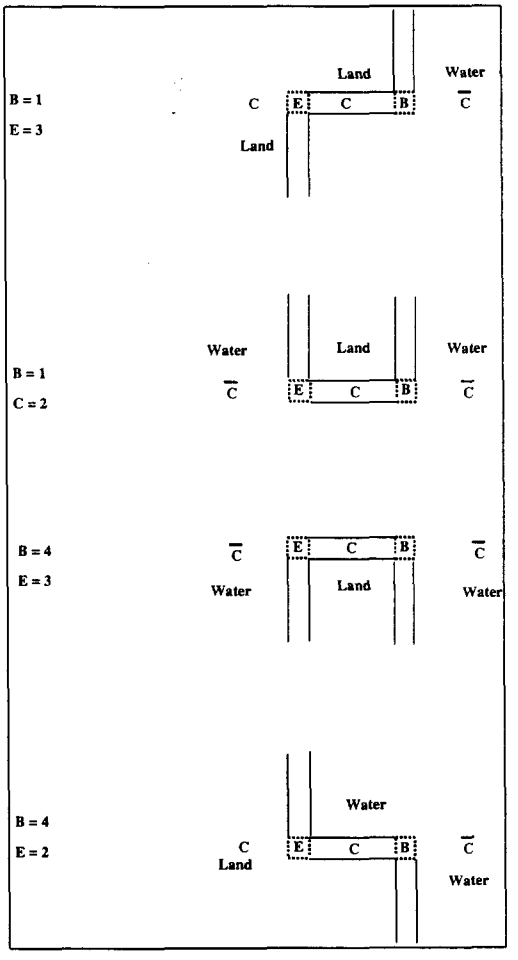

Figure 4: Assuming scanning from right to left, this figure illustrates the four possible cases for horizontal runs with water on the right of the horizontal boundary run.

B	E	Clear Pixel State
1	3	Toggle
1	2	Don't Toggle
4	3	Don't Toggle
4	2	Toggle

In the table we see that if the $|B - E| = 2$, the clear pixel value should be toggled. If $|B - E| = 1$, the clear pixel value should stay the same before and after the horizontal run.

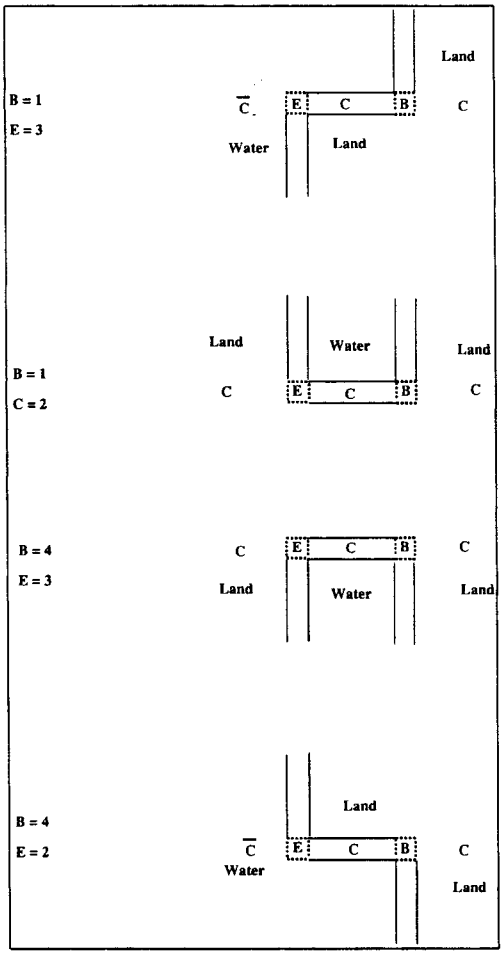

Figure 5: Assuming scanning from right to left, this figure illustrates the four possible cases for horizontal runs with land on the right of the horizontal boundary run.

The 8 cases for scanning from left to right are illustrated in Figures 6 and 7. In these figures, **C** indicates the clearing pixels state (i.e. pixels should be cleared), and $\overline{\mathbf{C}}$ indicates the state where pixels aren't cleared.

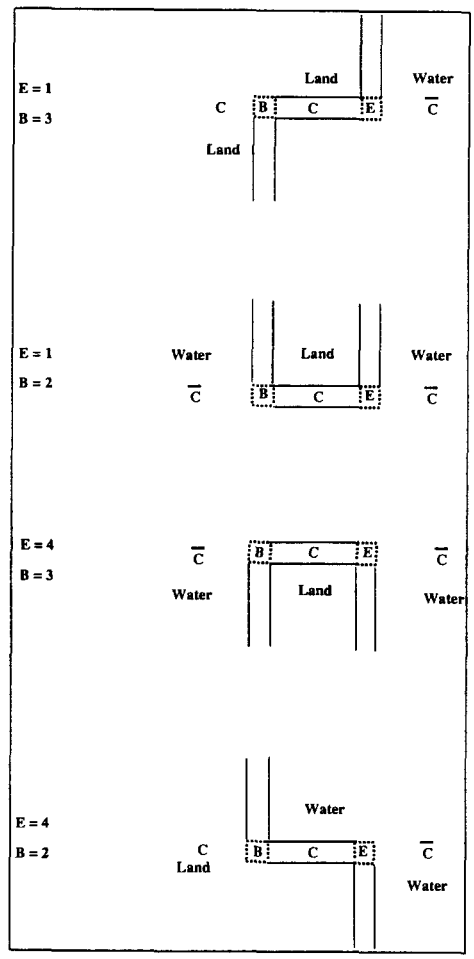

Figure 6: Assuming scanning from left to right, this figure illustrates the four possible cases for horizontal runs with water on the right of the horizontal boundary run.

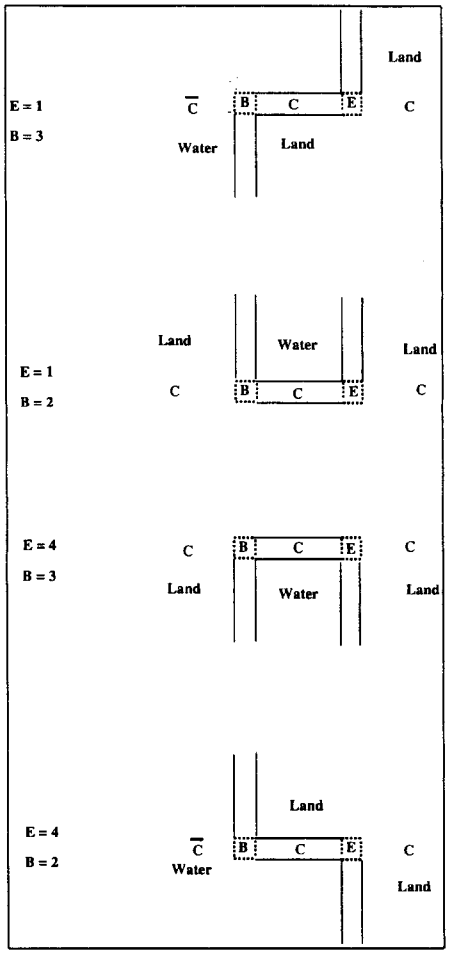

Figure 7: Assuming scanning from left to right, this figure illustrates the four possible cases for horizontal runs with land on the right of the horizontal boundary run.

B	E	Clear Pixel State
3	1	Toggle
2	1	Don't Toggle
3	4	Don't Toggle
2	4	Toggle

Once again we see that if the $|B - E| = 2$, the clear pixel value should be toggled. If $|B - E| = 1$, the clear pixel value should stay the same before and after the horizontal run.

4 Results

The algorithm for automatic coastline detection was tested on the image shown in Figure 8. This image has 12592 rows and 2000 columns. There are two final results: (1) the original image with the coastline overlayed; and (2) the coastline, itself, with the islands shown. The large size of the image make it difficult to display the single pixel coastline overlayed on the original image (the single pixel becomes too fine to see). Figure 9 shows the single pixel coastline that is output from the algorithm, along with the detected islands.

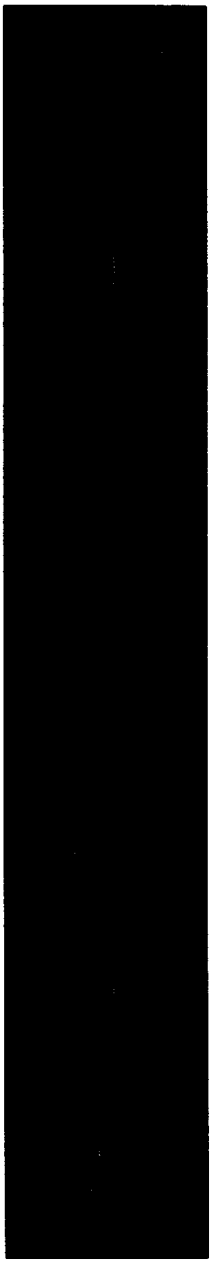

Figure 8: Original large coastline analyzed.

Figure 9: Boundary for Original large coastline analyzed.

In order to better understand the algorithm, it is helpful to examine intermediate results. In order to better illustrate the actions of the algorithms, we will work will a smaller subset of the large coastline. Figure 10 is a 1999 by 1999 subset taken from Figure 8. Results from each of the intermediate steps done on this smaller image are shown in subsequent figures.

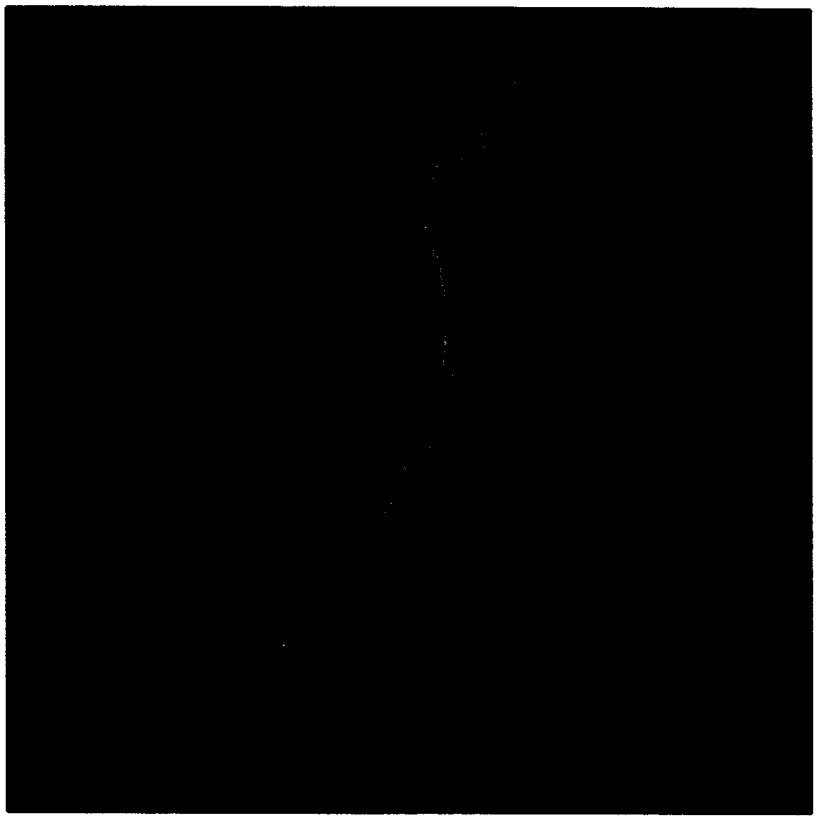

Figure 10: Original 1999 x 1999 coastline analyzed.

Figure 11 is the image after two passes through the 3x3 median filter. After the median filtering, the variations in the water and land are reduced. There are some additional bright pixels added, but the overall speckling is reduced. There are still some residual areas of moderate reflectance in the water, especially near the shoreline. Subsequent passes through the median filter would help to remove this, but at the cost of longer computation time.

Figure 12 is the image after median filtering and histogram flattening. The flattening brings out all the details in the water that we will need to remove. It is clear from this figure that in general the pixel values in water have a lower value than those in land.

Figure 13 is the result of threshholding the previous image at a value of 200. The threshholding level was chosen empirically. Because of the histogram equalization step, this value should be fairly robust. Depending on different imaging geometries or terrain (different land or water conditions), the actual threshhold value may need to be changed to give optimal performance.

The final image processing steps are two passes through a 7x7 dilation filter. Recall that these steps are necessary to close any gaps in the land/water boundary. The results of the dilation filtering are shown in Figure 14. The effect of the dilation is to make the bright parts of the images grow. As a result of the dilation, the

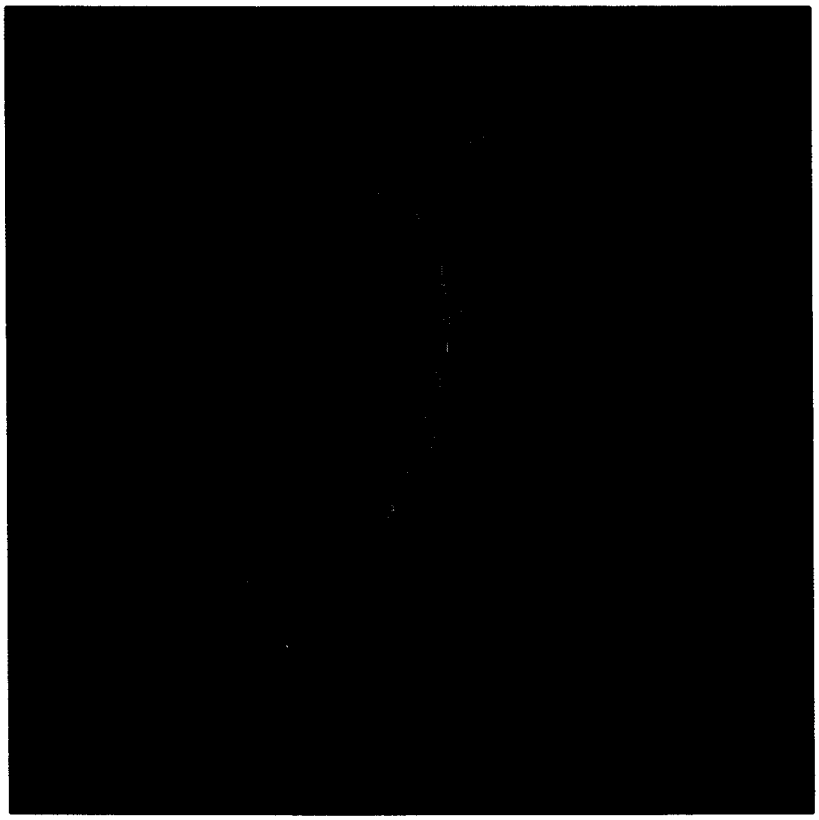

Figure 11: Results after two passes with a 3x3 median filter.

border does move out slightly.

After the image processing steps are completed, the coastline is identified using the countour following algorithm. Figure 15 shows the detected coastline overlayed on the original image. As discussed previously, the detected coastline is continuous and one pixel thick.

Figure 16 is the detected coastline and the identified islands. The coastline for the islands are not automatically detected. The main purpose is to indicate the presence of islands to an analyst. The analyst can then determine if the islands are significant enough to warrant recalculation of the coastline. Further modifications to the software can be added that can handle certain conditions automatically.

Figure 12: Results after median filtering and histogram equalization.

Figure 13: Results after median filtering and histogram equalization and threshholding at 200.

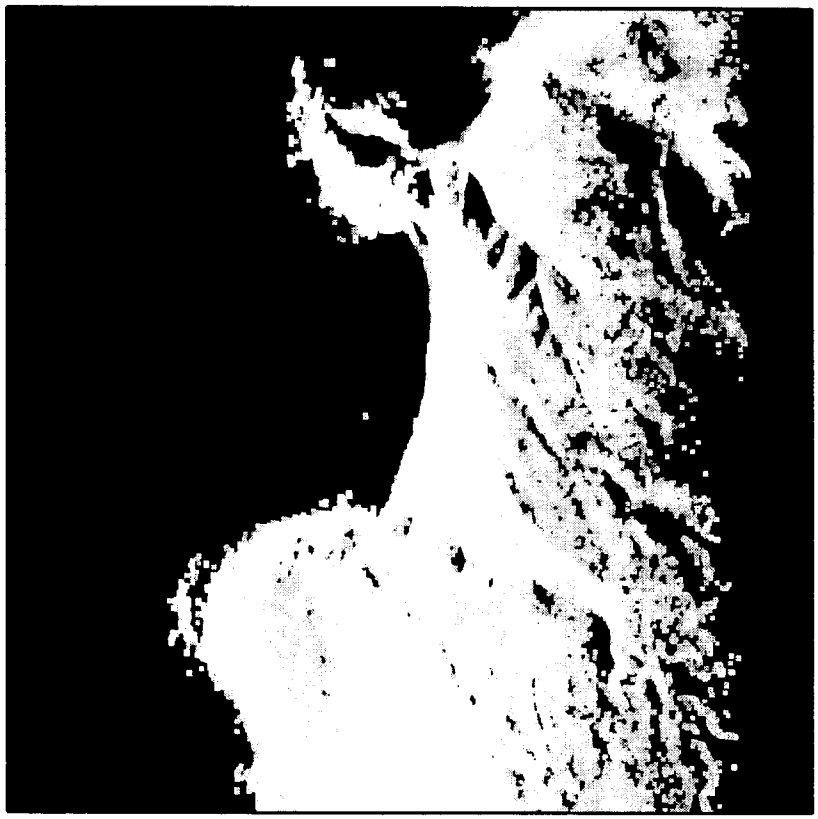

Figure 14: Results after median filtering, histogram equalization, threshholding and two passes of a 7x7 dilation operation.

Figure 15: The final result: the automatically detected coastline overlayed over the original image.

Figure 16: The automatically detected coastline and the detected islands.

The image we have been studying has some interesting features. Note that in Figure 10, the upper peninsula has a high cliff, which causes a shadow in the land. This is a rather difficult situation for the algorithm to handle, because the shadow from the cliff is so close to the shore. Here the dilation step is critical. Without it, the countour following algorithm would have "walked" inland and made the shadow into a bay.

In Figure 10, there are various isolated bright pixels near the shoreline. We do not have ground truth, but these bright returns could be due to either rocky jetties, or due to ice chunks in the water. The image processing steps attempt to minimize the smaller isolated returns that could be small ice chunks, while maintaining larger, brighter clusters. The image processing steps also try to cluster with the mainland bright pixels very close to the shoreline. The contour following then follows the mainland coastline, but any significant isolated clusters in the water are marked.

Another interesting feature is shown in Figure 17. Near the shoreline there is a very smooth light gray border. Once again we do not have ground truth, so it is hard to know if the image shown in Figure 17 contains a shallow beach with water covering some of the sand, or if it is an artificial artifact of the image formation. This area needs to be ground truthed, in order to see how accurately the algorithm performed.

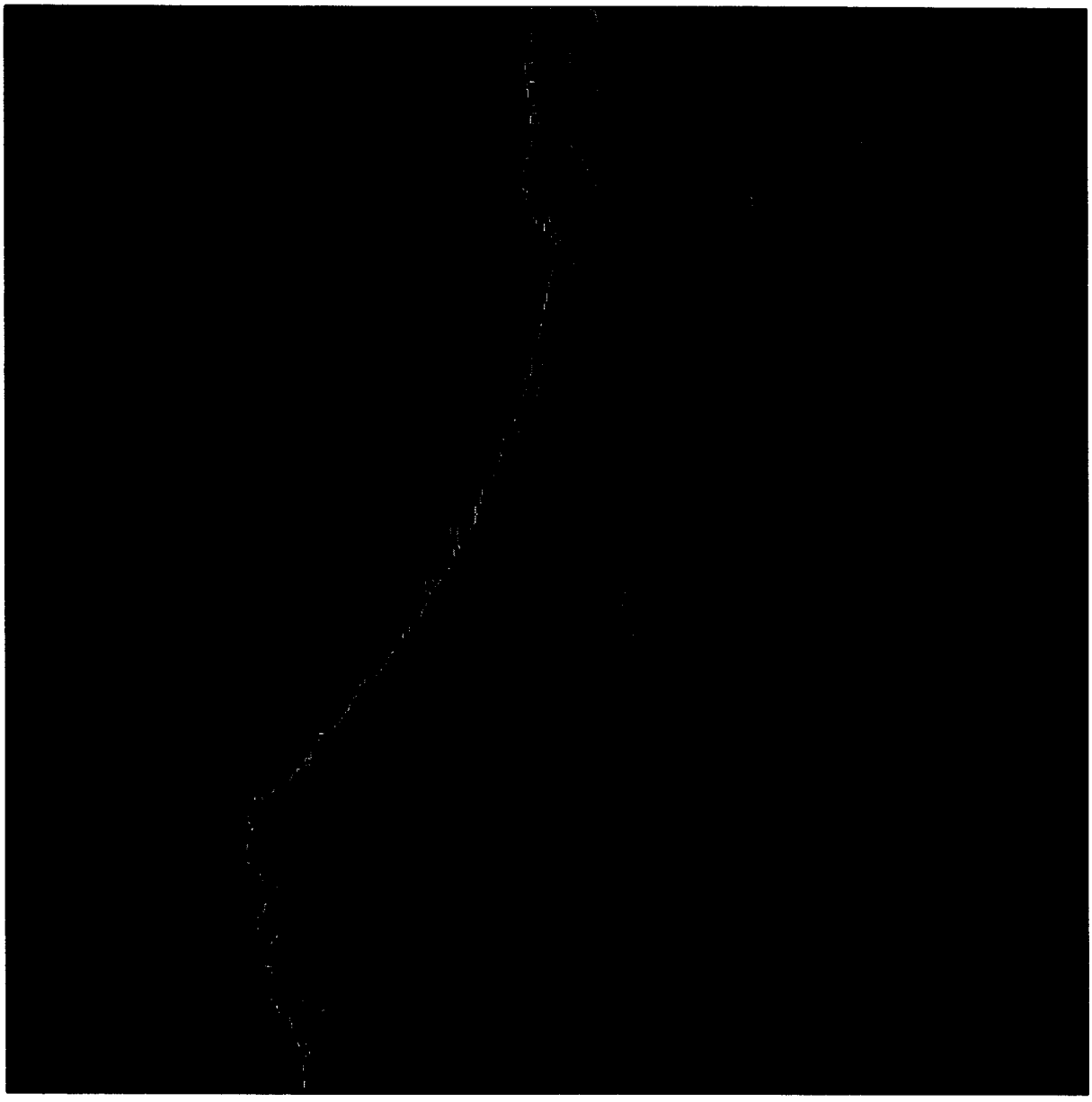

Figure 17: Subset of the large coastline including a beach. The detected coastline is overlayed on the original image.

5 Conclusions

In this report we have presented an automatic coastline detector for SAR imagery. The algorithm requires some image pre-processing before the coastline detection step is run. The results from the algorithm are a single pixel boundary between land and water, and the identification of islands. The algorithm appears to be robust, but more testing is necessary. Also, many variations and additions to the algorithm can be made, as required by a particular application.

6 Future Work

This report simply documents a first-cut algorithm directed at automatic coastline detection in SAR images. Future work to this algorithm includes further assessment of robustness and addressing specific requirements for specific applications. These might be such things as automatically finding island boundaries, in addition to just identifying the islands, and altering the start and stop conditions perhaps to guarantee finding a mainland/water boundary. Future work also includes transforming the detected boundary to standard earth coordinates.

A Example of How to Run Automatic Coastline Detection Code

This appendix shows the sequence of programs to run to produce the automatically detected coastline and overlays. The programs are assumed to be in a directory /CODE that is at the same level as a directory containting the data. The original data is a one byte detected image, with suffix .s1. All the image processing code works on dsp files with a .dsp suffix. The .i suffix refers to images without header and in a short int format. saoimage is a UNIX tool for displaying images.

Following is a brief description of the various codes. The codes use the image processing library supplied with Reference [EK91]. Additional programs were written to implement the automatic coastline detection. These additional programs that are not part of the image processing library supplied with Reference [EK91] are included in subsequent appendices.

extractsubsetDSP is a program that extracts an image subset from a one byte image. From the subset, it produces a dsp file and a short int image file.

nonlin2dsize is a program that performs multiple types of nonlinear filtering: erosion, dilation or median filtering. This program takes user input to determine the type of filtering, the window size and the number of passes.

flatten is a program that performs the histogram equalization on an image.

threshhold is a program that thresholds an image. It takes a threshold value as user input.

followboundarytopologyV2 is the code that includes the contour following. It takes as input the preprocessed image, and returns a file containing the coastline and islands.

overlay is a program that takes the boundary found in followboundarytopologyV2 and overlays it over an image.

removehdr is a program that strips off the header from a dsp file. The result is a short int image file that can be used in a number of display programs.

```
> ../CODE/extract/extractsubsetDSP rma_out.2048.16384.s1 test.dsp test.i
Enter the number of rows to skip
4000
Enter the number of columns to skip
0
Enter the number of rows Input
16384
Enter the number of columns Input
2048
Enter the number of rows Output
1999
Enter the number of columns Output
1999
Size of short int = 2
> saoimage -ul -i2 1999 1999 test.i &

> cd TEST/
/home/iaertez/ALASKA/TEST
test.dsp        test.i
> ../CODE/nonlin2dsize

Enter name of input image file : test.dsp
Enter the function desired:
  1 -- Erosion of light areas (min)
  2 -- Dilation of light areas (max)
  3 -- Median filter of the image

Enter your choice [1...3] : 3

Enter number of passes [1...10] : 2

Enter value of size for kernel [3...15] : 3

   0  16  32  48  64  80  96 112 128 144 160 176 ...
   0  16  32  48  64  80  96 112 128 144 160 176 ...
```

```
Enter filtered output file name : test.med3x2.dsp
> ls
test.dsp        test.i          test.med3x2.dsp
>
> ../CODE/flatten

Enter image file to be histogram flattened : test.med3x2.dsp

Enter histogram file name : test.hst

Enter histogram flattened image name : test.med3x2.flt.dsp

> ls
test.dsp                test.i                  test.med3x2.flt.dsp
test.hst                test.med3x2.dsp

> ../CODE/threshhold

Enter image file to be threshholded  : test.med3x2.flt.dsp

Enter value of detection threshold [0...255] : 200

Enter threshholded image name : test.med3x2.flt.thr.dsp
>
> ../CODE/nonlin2dsize

Enter name of input image file : test.med3x2.flt.thr.dsp
Enter the function desired:
  1 -- Erosion of light areas (min)
  2 -- Dilation of light areas (max)
  3 -- Median filter of the image

Enter your choice [1...3] : 2

Enter number of passes [1...10] : 2

Enter value of size for kernel [3...15] : 7

   0  16  32  48  64  80  96 112 128 144 160 176 ...
   0  16  32  48  64  80  96 112 128 144 160 176 ...
Enter filtered output file name : test.med3x2.flt.thr.dil7x2.dsp
>
../CODE/followboundarytopologyV2

Enter image file to find boundary in  : test.med3x2.flt.thr.dil7x2.dsp

Enter What side of the image is the land.  left==> -1; right ==> 1 [-1...1] : 1
```

```
Enter Set threshhold to use as criteria for land [0...255] : 200

Enter Do you want to input the row to start coastline search?
                    yes==>1; no==> 0 [0...1] : 1

Enter Enter the row number to look for starting point [0...1990] : 1987

Enter Enter the row number to stop searching for coastline [10...1990] : 10

Enter boundary image name : test.bndry.i
>


> ../CODE/extract/removehdr test.med3x2.flt.thr.dil7x2.dsp test.med3x2.flt.thr.dil7
Number of Records (rows) 1999
Number of entries per record (cols)1999
Data type (code )5
Data size (size of each element)2

> saoimage -ul -i2 1999 1999 test.med3x2.flt.thr.dil7x2.i

>  ../CODE/overlay

Enter image file to overlay boundary on : test.dsp

Enter image boundary file to be overlayed  : test.bndry.dsp

Enter overlayed image name : test.overlay.dsp
>
> ../CODE/extract/removehdr test.overlay.dsp test.overlay.i
Number of Records (rows) 1999
Number of entries per record (cols)1999
Data type (code )5
Data size (size of each element)2
> saoimage -ul -i2 1999 1999 test.overlay.i &
```

B extractsubsetDSP.c

```
#include "include.h"   /* This declares all the global vars as EXTERN to this file */
#include "global.h"    /* This declares all the global vars  */
#include "defines.h"   /* This includes constant and macro defs*/

typedef struct {
    unsigned char type; /* data type */
    unsigned char element_size; /* size of each element of data */
    unsigned short int records; /* number of data records */
    unsigned short int rec_length; /* number of elements in each record */
    } HEADER;

main(int argc, char **argv)
{
```

```
/*************************************************************
This program extracts a subset from a .s1 (one byte char image)
and makes it into a short int DSP file.
*************************************************************/

/*************************************************************
************** VARIABLE DECLARATIONS*********************
*************************************************************/
    int   i, iRowskip, iColskip,
          iNumRowsIn, iNumColsIn  ,
          iNumRowsOut, iNumColsOut,
          uintsize, j;
    unsigned char *puiBuffer;
    short int *psiBuffer;
    FILE  *filepInput, *filepOutputI, *filepOutputDSP;
    HEADER *headerinfo;

    headerinfo = (HEADER *) malloc(sizeof(HEADER)*1);
    if((filepInput=fopen(argv[1],"r"))==NULL)
    {
        printf("Unable to open input file %s\n", argv[1]);exit(0);
    }

    if((filepOutputDSP=fopen(argv[2],"w"))==NULL)
            { printf("Unable to open output file %s \n", argv[2]);exit(0);}

    if((filepOutputI=fopen(argv[3],"w"))==NULL)
            { printf("Unable to open output file %s \n", argv[3]);exit(0);}

    printf("Enter the number of rows to skip \n");
    scanf("%d", &iRowskip);

    printf("Enter the number of columns to skip \n");
    scanf("%d", &iColskip);

    printf("Enter the number of rows Input \n");
    scanf("%d", &iNumRowsIn);

    printf("Enter the number of columns Input\n");
    scanf("%d", &iNumColsIn);

    printf("Enter the number of rows Output\n");
    scanf("%d", &iNumRowsOut);

    printf("Enter the number of columns Output \n");
    scanf("%d", &iNumColsOut);

    printf("Size of short int = %d \n", sizeof(short int));
    uintsize = sizeof (unsigned char);
    puiBuffer = (unsigned char  *)malloc(sizeof(unsigned char ) * iNumColsOut);

    psiBuffer = (short int *)malloc(sizeof(short  int ) * iNumColsOut);

    i=5;  /* signed integer */
    headerinfo->records = (unsigned short int ) iNumRowsOut;
    headerinfo->type = (unsigned char ) i; /* signed int */
    headerinfo->rec_length= (unsigned short int ) iNumColsOut;
    headerinfo->element_size= sizeof (short int);

    fwrite(headerinfo,sizeof(HEADER),  1,filepOutputDSP);

    for (i=0; i<iNumRowsOut; i++)
    {
         /* fseek to correct row*/
        fseek(filepInput, (iRowskip+i)*iNumColsIn  * uintsize ,0);
         /* fseek to correct range sample*/
        fseek(filepInput, iColskip*  uintsize, 1);
         /* read in the number of samples to be processed */
        fread(puiBuffer,uintsize  , iNumColsOut  ,filepInput);
        for (j=0; j<iNumColsOut; j++)
        {
           *(psiBuffer+j)= (short int ) *(puiBuffer+j);
        }
        fwrite(psiBuffer,sizeof(short  int), iNumColsOut,filepOutputDSP);
        fwrite(psiBuffer,sizeof(short  int), iNumColsOut,filepOutputI);
    }

}
```

C removehdr.c

```
#include "include.h"    /* This declares all the global vars as EXTERN to this file */
#include "global.h"    /* This declares all the global vars  */
#include "defines.h"    /* This includes constant and macro defs*/
```

```
typedef struct {
    unsigned char type; /* data type */
    unsigned char element_size; /* size of each element of data */
    unsigned short int records; /* number of data records */
    unsigned short int rec_length; /* number of elements in each record */
    } HEADER;

main(int argc, char **argv)
{
/************************************************************
************** VARIABLE DECLARATIONS*********************
************************************************************/
    int   i,
          j;
    short int    *piBuffer;
    FILE  *filepInput, *filepOutputI ;
    HEADER *headerinfo;

    headerinfo = (HEADER *) malloc(sizeof(HEADER)* 1);
    if((filepInput=fopen(argv [1],"r"))==NULL)
    {
        printf("Unable to open input file %s\n", argv[1]);exit(0);
    }

    if((filepOutputI=fopen(argv[ 2],"w"))==NULL)
             { printf("Unable to open output file %s \n", argv[3]);exit(0);}


    fread(headerinfo,sizeof(HEADER),  1,filepInput);
    printf("Number of Records (rows) %d \n", headerinfo->records);
    printf("Number of entries per record (cols)%d \n", headerinfo->rec_length);
    printf("Data type (code )%d \n", (int) headerinfo->type);
    printf("Data size (size of each element)%d \n", (int) headerinfo->element_size);

    piBuffer = (short int *)malloc(sizeof(short  int ) * headerinfo->rec_length);

    for (i=0; i<(int ) headerinfo->records;  i++)
    {
         /* read in the number of samples to be processed */
        fread(piBuffer,sizeof(short  int) , headerinfo->rec_length,filepInput);
        fwrite(piBuffer,  sizeof(short  int), headerinfo->rec_length,filepOutputI);
    }

}
```

D nonlin2dsize.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "get.h"

/************************************************************

NONLIN2D: This program does min, max, or median filtering
          of an image using the function nonlin2d().

INPUTS: DSP format image file.

OUTPUTS: DSP format image file.

************************************************************/

main()
{
  MATRIX   *IN, *OUT;
  int      i, passes, inval, filtype, size;

  do{
    IN = matrix_read(get_string("name  of input image file"));
  }while(!IN);

  printf("Enter the function desired:\n");
  printf("  1 -- Erosion of light areas (min)\n");
  printf("  2 -- Dilation of light areas (max)\n");
  printf("  3 -- Median filter of the image\n");

  inval = get_int("your choice", 1, 3);

  passes = get_int("number of passes", 1, 10);

  filtype = inval - 1;

/* Get the size value as a user input */
```

```
  size = get_int("value of size for kernel", 3, 15);

  for (i=0; i<passes; i++){
    OUT = nonlin2d(IN, size, filtype);
    i++;
    if (i < passes)
      IN = nonlin2d(OUT, size, filtype);
  }

  if (passes%2)
    matrix_write(OUT, get_string("filtered output file name"));
  else
    matrix_write(IN, get_string("filtered output file name"));
}
```

E threshhold.c

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "get.h"
#include  "matrix.h"

/*************************************************************

Threshhold :  THIS PROGRAM  will threshhold the image
          at a value specified by the user.

INPUTS:    IMAGE TO BE threshholded IN DSP FILE FORMAT

OUTPUTS:   NEW Threshholded IMAGE .

*************************************************************/

main()
{
  DSP_FILE   *dsp_info;
  MATRIX     *IN, *OUT;
  int        i, j, temp_int, min, max, threshold;
  short int *in, *out;
  char       *in_name, trail[100];

/* Read input file into a matrix structure.  */
  do{
    in_name = get_string("image file to be threshholded ");
    IN = matrix_read(in_name);
  }while(!IN);

  if (IN->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }

 min = 0;
 max = 255;
/* Get the threshold value as a user input */
  threshold = get_int("value of detection threshold", min, max);


/* Using the threshhold create a new image
   from the original */
  OUT = matrix_allocate(IN->rows, IN->cols, sizeof(short int));

  for (i=0; i<IN->rows; i++){
    in  = (short int *)IN->ptr[i];
    out = (short int *)OUT->ptr[i];
    for (j=0; j<IN->cols; j++){
      temp_int = in[j];
      temp_int = temp_int>255 ? 255 : temp_int;
      temp_int = temp_int<threshold   ?   0 : temp_int;
      out[j] =  (short int) temp_int;
    }
  }


/* Write the new image to disk */
  matrix_write(OUT, get_string("threshholded image name"));
}
```

F followboundarytopologyV2.c

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "get.h"
#include  "matrix.h"

/*************************************************************

followboundary:  THIS PROGRAM  will follow the boundary in
          an image.  The user must specify which half of
          the image represents the object (land).  A threshhold
          value will also be input by the user to designate
          a pixel level criteria to represent the object.

INPUTS:   IMAGE TO BE threshholded IN DSP FILE FORMAT
          Direction (i.e.  which side of the image is the
                     object on )
          Threshhold to determine what is an object pixel

This modification (V2) asks the user for a start row,
and also exits gracefully if the coastline falls off the
image at either side or bottom of image.

OUTPUTS:  NEW boundary IMAGE .

*************************************************************/

main()
{
  DSP_FILE  *dsp_info;
  MATRIX    *IN, *OUT;
  int       i, j, temp_int, min, max, DIR, threshold, orientation,
            reachedend, clearpixels, cont, prevpix, toggle, q, val, horizbound,
            below,belowleft, belowright, above, aboveleft, aboveright,
            BeginTopo, EndTopo, BeginState, starti, startj, userinput ,
            mini, exitrow;
  short int *in, *out, *inbelow, *inabove;
  char      *in_name, trail[100];

/* Read input file into a matrix structure.  */
  do{
    in_name = get_string("image file to find boundary in ");
    IN = matrix_read(in_name);
  }while(!IN);

  if (IN->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }

 min = -1;
 max = 1;
/* Get the direction that the boundary finder should travel.
   If the land is on the right side of the image, the boundary
   finder will travel clockwise, and DIR=1;
   If the land is on the left side of the image, the boundary
   finder will travel counter clockwise and DIR = -1;
*/
  DIR = get_int("What side of the image is the land.  left==> -1; right ==> 1", min, max);
  min = 0;
  max = 255;
  threshold = get_int("Set threshhold to use as criteria for land", min, max);

   starti = 0;
   startj = 0;
   userinput = get_int("Do you want to input the row to start coastline search?  yes==>1; no==> 0" , 0, 1);
   if (userinput==1)
   {
      starti = get_int("Enter the row number to look for starting point",
                min, IN->rows -9);
      i = starti;
      exitrow = get_int("Enter the row number to stop searching for coastline",
                10, IN->rows -9);
   }
   else
   {
   i = IN->rows -9;  /* bottom most row */
   starti = i;
   }


/* Scan the bottom row input image  for the first land pixel.
   For land on right, dir=1, start scanning from the left.
   For land on left , dir=-1, start scanning from the right.
*/

   in  = (short int *)IN->ptr[i];
   if (DIR == 1)
```

```
   {
      j = 0;
      while ((int )in[j] < threshold)
      {
        printf("i,j = %d, %d, val = %d  \n", i, j, in[j]);
        j++;
        if (j==IM->cols-1)
        {
          printf("No starting point found on row %d \n", starti);
          exit;
        }
      }
   }
   else if (DIR == -1)
   {
      j = IM->cols -1;
      while ((int )in[j] < threshold)
      {
        j--;
        if (j==0)
        {
          printf("No starting point found on row %d \n", starti);
          exit;
        }
      }
   }

/*
    Start pixel is [i, (j-DIR)]
    Start orientation is 90 + DIR*90
*/
   j = j  - DIR;
   orientation = 90 + 90*DIR;
   startj = j;
   mini = starti;

   printf("Starting pixel is (%d, %d) \n", i, j);

   reachedend = 0;

   do
   {
    in  = (short int *)IM->ptr[i];
/*
    cont= get_int("Continue the search", min, max);
*/
    printf("\n in[%d, %d] = %d \n", i, j, (int) in[j]);
    printf("orientation = %d \n ", orientation);

    mini = i< mini? i  : mini;

    if (i <exitrow)
    {
        printf("Reached top (i = %d ) \n", i);
        while   (in[j] <threshold)
        {
           j = j + DIR;
        }
        in[j]= -1;
        reachedend = 1;
    }
    else if ( (abs(i-starti) < 1) && ((starti - mini)> 15) )
    {
        printf("Reached bottom again (i = %d) \n", i);
        reachedend = 1;
    }
    else if ( (j==0) && ((int )in[j]>=threshold) )
    {
        in[j] = -1;
        orientation = 0;
        i=i-1;
    }
    else if ( (j==IM->cols -1) && ((int )in[j]>=threshold) )
    {
        in[j] = -1;
        printf("Marked pixel is (%d, %d) \n", i, j);
        orientation = 180;
        i=i-1;
    }
    else if ( ((int )in[j] >=threshold)  || ((int )in[j] == -1) )
    {
       in[j] = -1;
       switch (orientation)
       {
          case 0:
             printf("Case 0 Land  \n");
             i = i-DIR;
             break;
          case 270:
             printf("Case 270 Land \n");
             j = j+DIR;
             break;
          case 180:
```

```
            printf("Case 180 Land \n");
            i = i+DIR;
            break;
         case 90:
            printf("Case 90 Land \n");
            j = j-DIR;
            break;
      }
      orientation =
           (orientation + 90*DIR) < 360 ? (orientation + 90*DIR): (orientation + 90*DIR)-360;
   }
   else
   {
      switch (orientation)
      {
         case 0:
            printf("Case 0 \n");
            i = i+DIR;
            break;
         case 270:
            printf("Case 270 \n");
            j= j-DIR;
            break;
         case 180:
            printf("Case 180 \n");
            i= i-DIR;
            break;
         case 90:
            printf("Case 90 \n");
            j= j+DIR;
            break;
      }
      orientation =
           (orientation - 90*DIR) >= 0 ? (orientation - 90*DIR): (orientation - 90*DIR)+360;
   }
  }
  while (reachedend==0);

  printf(" Finished marking pixels \n ");

 OUT = matrix_allocate(IN->rows, IN->cols, sizeof(short int));

 if (DIR == 1)
 {

/* LATER, COPY THE IMAGE BOUNDARY ROWS AT BORDERS WHICH AREN'T SCANNED */

     for (i=1; i<IN->rows-1; i++)
     {
       clearpixels = 1;
       prevpix = 0;
       horizbound = 0;
       in  = (short int *)IN->ptr[i];
       inbelow= (short int *)IN->ptr[i+1];
       inabove= (short int *)IN->ptr[i-1];
       out = (short int *)OUT->ptr[i];

       for ( j=IN->cols-2; j >=0;  j--)
       {
         if (clearpixels ==1)
         {
            temp_int = in[j];
            if (temp_int != -1)
            {
               temp_int = 0;
               if (horizbound ==1)
               {
                   horizbound =0;

                  /* figure out end topo */
                   below = inbelow[j+1];
                   belowleft = inbelow[j];
                   above = inabove[j+1];
                   aboveleft = inabove[j];
                   if ( (below==-1) || (belowleft == -1) )
                   {
                      EndTopo = 3;
                   }
                   else if ( (above ==-1) || (aboveleft == -1) )
                   {
                      EndTopo = 2;
                   }
                  /* Depending on beginning and end topo,
                     figure out state for clearpixels */
                   toggle = BeginTopo - EndTopo;
                   toggle = toggle > 0? toggle: -1*toggle;
                   if (toggle ==1) clearpixels = BeginState;
                   else if (toggle ==2) clearpixels = -1 * BeginState;
               }
               prevpix = 0;
            }
            if (temp_int == -1)
            {
```

```
            temp_int = 255;
            if (prevpix== 0)
            {
                below = inbelow[j];
                belowright= inbelow[j+1];
                above = inabove[j];
                aboveright= inabove[j+1];
                if ( (above==-1) || (aboveright == -1) )
                {
                   BeginTopo = 1;
                }
                else if ( (below ==-1) || (belowright == -1) )
                {
                   BeginTopo = 4;
                }
                BeginState = clearpixels;


                clearpixels = clearpixels * -1;
            }
            if (prevpix== -1)
            {
               horizbound = 1;
               clearpixels = 1;
            }
            prevpix= -1;
          }
          out[j] =  (short int) temp_int;
        }
        else  /* clearpixels = -1 */
        {
           temp_int = in[j];
           if (temp_int != -1)
           {
              prevpix= 0;
        /*      clearpixels = 1; */
           }
           temp_int = temp_int>255 ? 255 : temp_int;
           if (temp_int == -1)
           {
              temp_int = 255;
              if (prevpix== 0)
              {
                  below = inbelow[j];
                  belowright= inbelow[j+1];
                  above = inabove[j];
                  aboveright= inabove[j+1];
                  if ( (above==-1) || (aboveright == -1) )
                  {
                     BeginTopo = 1;
                  }
                  else if ( (below ==-1) || (belowright == -1) )
                  {
                     BeginTopo = 4;
                  }
                  BeginState = clearpixels;


                  clearpixels = clearpixels * -1;
              }
              if (prevpix== -1)
              {
                 horizbound = 1;
                 clearpixels = 1;
              }
              prevpix= -1;
           }
           temp_int = temp_int<threshold   ?   0 : temp_int;
           out[j] =  (short int) temp_int;
        }
      } /* for each column */
    } /* for each row */
  }   /* DIR = 1 */
  else if (DIR == -1)
  {

/* LATER, COPY THE IMAGE BOUNDARY ROWS AT BORDERS WHICH AREN'T SCANNED */

      for (i=1; i<IN->rows-1; i++)
      {
        clearpixels = 1;
        prevpix = 0;
        horizbound = 0;
        in  = (short int *)IN->ptr[i];
        inbelow= (short int *)IN->ptr[i+1];
        inabove= (short int *)IN->ptr[i-1];
        out = (short int *)OUT->ptr[i];

        for ( j = 1; j< IN->cols-1; j++)
        {
          if (clearpixels ==1)
          {
             temp_int = in[j];
```

```
   if (temp_int != -1)
   {
      temp_int = 0;
      if (horizbound ==1)
      {
          horizbound =0;

         /* figure out end topo */
          below = inbelow[j-1];
          belowright= inbelow[j];
          above = inabove[j-1];
          aboveright= inabove[j];
          if ( (above==-1) || (aboveright == -1) )
          {
             EndTopo = 1;
          }
          else if ( (below ==-1) || (belowright == -1) )
          {
             EndTopo = 4;
          }

         /* Depending on beginning and end topo,
            figure out state for clearpixels */
          toggle = BeginTopo - EndTopo;
          toggle = toggle > 0? toggle: -1*toggle;
          if (toggle ==1) clearpixels = BeginState;
          else if (toggle ==2) clearpixels = -1 * BeginState;
      }
      prevpix = 0;
   }
   if (temp_int == -1)
   {
      temp_int = 255;
      if (prevpix== 0)
      {
          below = inbelow[j];
          belowleft = inbelow[j-1];
          above = inabove[j];
          aboveleft = inabove[j-1];
          if ( (below==-1) || (belowleft == -1) )
          {
             BeginTopo = 3;
          }
          else if ( (above ==-1) || (aboveleft == -1) )
          {
             BeginTopo = 2;
          }

          BeginState = clearpixels;

          clearpixels = clearpixels * -1;
      }
      if (prevpix== -1)
      {
         horizbound = 1;
         clearpixels = 1;
      }
      prevpix= -1;
   }
   out[j] =  (short int) temp_int;
}
else  /* clearpixels = -1 */
{
   temp_int = in[j];
   if (temp_int != -1)
   {
      prevpix= 0;
/*      clearpixels = 1; */
   }
   temp_int = temp_int>255 ? 255 : temp_int;
   if (temp_int == -1)
   {
      temp_int = 255;
      if (prevpix== 0)
      {
          below = inbelow[j];
          belowleft = inbelow[j-1];
          above = inabove[j];
          aboveleft = inabove[j-1];
          if ( (below==-1) || (belowleft == -1) )
          {
             BeginTopo = 3;
          }
          else if ( (above ==-1) || (aboveleft == -1) )
          {
             BeginTopo = 2;
          }

          BeginState = clearpixels;


          clearpixels = clearpixels * -1;
      }
```

```
            if (prevpix== -1)
            {
               horizbound = 1;
               clearpixels = 1;
            }
            prevpix= -1;
          }
          temp_int = temp_int<threshold   ?   0 : temp_int;
          out[j] =  (short int) temp_int;
        }
      } /* for each column */
    } /* for each row */

  }   /* DIR = -1 */



/* Write the new image to disk */
  matrix_write(OUT, get_string("boundary image name"));
}
```

G overlay

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "get.h"
#include  "matrix.h"

/*************************************************************

overlay:  THIS PROGRAM  will overlay a boundary over the image

INPUTS:
             IMAGE with boundary TO BE overlayed IN DSP FILE FORMAT
             IMAGE in which to overlay boundary IN DSP FILE FORMAT

OUTPUTS:  NEW overlayed IMAGE .

*************************************************************/

main()
{
  DSP_FILE   *dsp_info;
  MATRIX     *IN, *OUT, *INB;
  int        i, j, temp_intb, temp_int, min, max, threshold;
  short int *in, *out, *inb;
  char       *in_name, *in_nameb, trail[100];

/* Read input file into a matrix structure.  */
  do{
    in_name = get_string("image file to overlay boundary on");
    IN = matrix_read(in_name);
  }while(!IN);

  if (IN->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }

/* Read input file into a matrix structure.  */
  do{
    in_nameb = get_string("image boundary file to be overlayed ");
    INB = matrix_read(in_nameb);
  }while(!INB);

  if (INB->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }


/* Using the threshhold create a new image
   from the original */
  OUT = matrix_allocate(IN->rows, IN->cols, sizeof(short int));

  for (i=0; i<IN->rows; i++){
    in  = (short int *)IN->ptr[i];
    inb  = (short int *)INB->ptr[i];
    out = (short int *)OUT->ptr[i];
    for (j=0; j<IN->cols; j++){
      temp_int = in[j];
      temp_intb = inb[j];
      temp_int = temp_intb==255 ? 255 : temp_int;
```

```
      out[j] =  (short int) temp_int;
    }
  }


/* Write the new image to disk */
  matrix_write(OUT, get_string("overlayed image name"));
}
```

H overlay5x5.c

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "get.h"
#include  "matrix.h"

/*************************************************************

overlay:  THIS PROGRAM  will overlay a boundary over the image
          The boundary will be overlayed with a 5x5 width.

INPUTS:
             IMAGE with boundary TO BE overlayed IN DSP FILE FORMAT
             IMAGE in which to overlay boundary IN DSP FILE FORMAT

OUTPUTS:  NEW overlayed IMAGE .

*************************************************************/

main()
{
  DSP_FILE  *dsp_info;
  MATRIX    *IN, *OUT, *OUTTHICK, *INB;
  int       i, j, ii, jj, temp_intb, temp_int, min, max, threshold;
  short int *in, *out, *outthick, *inb;
  char      *in_name, *in_nameb, trail[100];

/* Read input file into a matrix structure.  */
  do{
    in_name = get_string("image file to overlay boundary on");
    IN = matrix_read(in_name);
  }while(!IN);

  if (IN->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }

/* Read input file into a matrix structure.  */
  do{
    in_nameb = get_string("image boundary file to be overlayed ");
    INB = matrix_read(in_nameb);
  }while(!INB);

  if (INB->element_size != sizeof(short int)){
    printf("\nError:  Input file is not of integer type\n");
    exit(1);
  }


/* Using the threshhold create a new image
   from the original */
  OUT = matrix_allocate(IN->rows, IN->cols, sizeof(short int));
 OUTTHICK = matrix_allocate(IN->rows, IN->cols, sizeof(short int));

  for (i=0; i<IN->rows; i++){
    in  = (short int *)IN->ptr[i];
    inb  = (short int *)INB->ptr[i];
    out = (short int *)OUT->ptr[i];
    outthick = (short int *)OUTTHICK->ptr[i];
    for (j=0; j<IN->cols; j++){
      temp_int = in[j];
      temp_intb = inb[j];
      temp_int = temp_intb==255 ? 255 : temp_int;
      out[j] =  (short int) temp_int;
      outthick[j] = out[j];
    }
  }


/* Write the new image to disk */
  matrix_write(OUT, get_string("overlayed image name"));
```

```
  for (i=5; i<IN->rows-5; i++)
    {
        out = (short int *)OUT->ptr[i];
        for (j=5; j<IN->cols-5; j++)
        {
            if (out[j] == 255)
            {
                for (ii = -5; ii <6; ii++)
                {
                    outthick = (short int *) OUTTHICK->ptr[i+ii];
                    for (jj = -5; jj < 6; jj++)
                    {
                        outthick[j+jj]= 255;
                    }
                }
            } /* end if */
        } /* end j*/
    } /* end i */
/* Write the new thick image to disk */
  matrix_write(OUTTHICK, get_string("overlayed THICK image name"));

}
```

I coastpreprocFinal.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "get.h"

/*************************************************************

coastpreproc : This program does  the sequency of image
          processing steps necessary to accentuate the land/
          water boundary for coastline detection.

          The steps are:
          - a 2 pass median filter to get rid of speckle
          - a histogram flattening
          - a thresholding step
          - a 2 pass dilation with kernal size of 7

          The parameters for these operations are fixed in
          the code (i.e. they are not input by the user).
          They may need to be twiddled with later.

INPUTS: DSP format image file.
        The image must be a short int file.

OUTPUTS: DSP format image file.
         This file will be the input to the coastline detector.

*************************************************************/

main()
{
  MATRIX   *IN, *OUT, *TEMP ;
  int      i, passes, inval, filtype, size;

/* variables from flatten */
  float     *hist_array;
  double    image_size, temp_flt;
  int        j, new_gray_level[256], temp_int;
  short int *in, *out;
  char      *in_name, trail[100];

/* variables from threshhold*/
  int       min, max, threshold;

/* First get in the name of the input file */
  do{
    IN = matrix_read(get_string("name  of input image file"));
  }while(!IN);

/* Our next step is to do a 10 pass median filter,
   so inval = 3 (determined by the nonlin2d function to
                 do median filter)
   and passes = 10
   Kernel size is 3
*/

  inval = 3;
  passes = 2;
  filtype = inval - 1;
```

```
  size = 3;  /* it can be between 3 and 15 */

/*
   Call the function nonlin2d to actually implement the median
   filter.  The output of nonlin2d is a pointer to the
   filtered image.
*/

   printf("MEDIAN FILTERING \n ");

  for (i=0; i<passes; i++){
    OUT = nonlin2d(IN, size, filtype);
    i++;
    if (i < passes)
      IN = nonlin2d(OUT, size, filtype);
  }

/* If you alter the number of passes to be other than 10,
the pointer to the result may be either IN or OUT.
If the number of passes is odd (passes%2 =1) the result
is in OUT.
If the number of passes is even (passes%2 =0) the result
is in IN.
*/

/* Now, we want to flatten out the median filtered image */

   printf("FLATTENING IMAGE \n ");


/**********************************************************

FLATTEN:  THIS PROGRAM USES THE FUNCTION histogram TO LEVEL
          THE HISTOGRAM OF AN INPUT IMAGE.

INPUTS:   IMAGE TO BE HISTOGRAM FLATTENED IN DSP FILE FORMAT

OUTPUTS:  NEW IMAGE WITH FLATTENED HISTOGRAM.

**********************************************************/


/* Our input file for the histogram is pointed to by the pointer
   IN */

/* Calculate the number of pixels in the image.  */
  image_size = (double) IN->rows * (double) IN->cols;

/* Create an array of 256 floats which represent the number
   of pixels in the image at a particular gray level.
   Write the array to disk as the first record in a
   two record DSP file. */

  hist_array = histogram(IN,0,255);

/*  The following commented section can bu used to write the histogram,
    if desired.  */

#ifdef WRITEHISTOGRAMS

   do{
    dsp_info = open_write(get_string("histogram  file name"),FLOAT,2,256);
  }while(!dsp_info);

  write_record((char *)hist_array,dsp_info);
#endif

/* Using the histogram array create a mapping of the original
   gray levels to the new histogram flattened gray levels */

  temp_flt = 0.;
  for(i=0; i<256; i++){     /* Loop thru original gray levels */

/* Find the distribution function of the image at each gray level */
    temp_flt += hist_array[i]/image_size;

/* Use the distribution function to create the mapping
   from old to new gray levels */
    new_gray_level[i] = (255.*temp_flt) + 0.5;
  }

/* Using the new mapping of gray levels create a new image
   from the original */
  OUT = matrix_allocate(IN->rows, IN->cols, sizeof(short int));

  for (i=0; i<IN->rows; i++){
    in  = (short int *)IN->ptr[i];
    out = (short int *)OUT->ptr[i];
    for (j=0; j<IN->cols; j++){
      temp_int = in[j];
      temp_int = temp_int>255 ? 255 : temp_int;
      temp_int = temp_int<0   ?   0 : temp_int;
      out[j] = new_gray_level[temp_int];
```

```
    }
  }

#ifdef WRITEHISTOGRAMS
/* Find the histogram of the newly created image and write as the
   second record in the file which is already open */

  hist_array = histogram(OUT,0,255);

  write_record((char *)hist_array,dsp_info);

/* make descriptive trailer for output histogram file */
  sprintf(trail,
    "Histograms of file %s before and after flattening",in_name);
  write_trailer(trail,dsp_info);
#endif

/* The flattened image is pointed to by OUT */

/* The next step is to threshhold the image. */

/* Our input file for the threshhold is pointed to by the pointer
   OUT */

   printf(" THRESHHOLDING IMAGE  \n ");

 min = 0;
 max = 255;
/*The threshhold we will use is 200.
This may need to be tweaked later .  The threshhold
should be between 0 and 255  */
  threshold = 200;

/* Using the threshhold create a new image from the original */

/* To make the terminology ok, let's assign the value of the pointer
OUT to  the pointer IN.  Then we can use the code in threshhold.c
as is.  */
  TEMP = IN;  /* save the location that IN pointed to, since
                 it was already allocated */
  IN = OUT;
  OUT = TEMP;


  for (i=0; i<IN->rows; i++){
    in  = (short int *)IN->ptr[i];
    out = (short int *)OUT->ptr[i];
    for (j=0; j<IN->cols; j++){
      temp_int = in[j];
      temp_int = temp_int>255 ? 255 : temp_int;
      temp_int = temp_int<threshold   ?   0 : temp_int;
      out[j] =  (short int) temp_int;
    }
  }

/* The threshholded image is pointed to by OUT */
/* To make the terminology ok, let's assign the value of the pointer
OUT to  the pointer IN.  Then we can use the code in nonlin2d.c
as is.  */
  TEMP = IN;  /* save the location that IN pointed to, since
                 it was already allocated */
  IN = OUT;
  OUT = TEMP;

/* The final step is to dilate the image with a kernel of 7
and with 2 passes.

   printf(" DILATING IMAGE  \n ");

   so inval = 2 (determined by the nonlin2d function to
                 do median filter)
   and passes = 4
   Kernel size is 5
*/

  inval = 2;
  passes = 2;
  filtype = inval - 1;
  size = 7;  /* it can be between 3 and 15 */

/*
   Call the function nonlin2d to actually implement the dilation
filter.  The output of nonlin2d is a pointer to the filtered image.
*/

  for (i=0; i<passes; i++){
    OUT = nonlin2d(IN, size, filtype);
    i++;
    if (i < passes)
      IN = nonlin2d(OUT, size, filtype);
  }

/* If you alter the number of passes to be other than 10,
```

```
the pointer to the result may be either IN or OUT.
If the number of passes is odd (passes%2 =1) the result
is in OUT.
If the number of passes is even (passes%2 =0) the result
is in IN.
*/

  if (passes%2)
     matrix_write(OUT, get_string("coast preprocessed file name"));
  else
     matrix_write(IN, get_string("coast preprocessed file name"));

} /* end main */
```

References

[BB82] Dana Ballard and Christopher Brown. *Computer Vision.* Prentice-Hall, Inc., 1982.

[DH73] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* J. Wiley and Sons, 1973.

[(Ed72] Satosi Watanabe (Editor). *Frontiers of Pattern Recognition.* Academic Press, 1972.

[(Ed88] J.C. Simon (Editor). *From Pixels to Features.* North-Holland, 1988.

[EK91] Paul Embree and Bruce Kimble. *C Language Algorithms for Digital Signal Processing.* Prentice Hall PTR, 1991.

[GW87] Rafael Gonzalez and Paul Wintz. *Digital Image Processing.* Addison-Wesley, 1987.

[Hor82] R. Michael Hord. *Digital Image Processing of Remotely Sensed Data.* Academic Press, 1982.

[Jah91] Bernd Jahne. *Digital Image Processing Concepts, Alorithms and Scientific Applications.* Springer-Verlag, 1991.

[RD84] Christian Ronse and Pierre Devijver. *Connected Components in Binary Images: the Detection Problem.* Research Studies Press, 1984.

[RK82] Azriel Rosenfeld and Avinash Kak. *Digital Picture Processing, Volumes 1 and 2.* Academic Press, second edition, 1982.

DISTRIBUTION:

5	MS 1207	I. A. Erteza, 5912
1	MS 1207	C. V. Jakowatz, 5912
1	MS 1207	D. A. Yocky, 5912
1	MS 9018	Central Technical Files, 8940-2
2	MS 0899	Technical Library, 4916
2	MS 0619	Review and Approval Desk, 12690 For DOE/OSTI